Exception Handling in Component-Based System Development

Alexander Romanovsky

University of Newcastle upon Tyne, UK

alexander.romanovsky@ncl.ac.uk

Designers of component-based software face two problems related to dealing with abnormal events: developing exception handling at the level of the integrated system and accommodating (and adjusting, if necessary) exceptions and exception handling provided by individual components. In this paper our intention is to develop an exception handling framework suitable for component-based system development by applying general exception handling mechanisms which have been proposed and successfully used in concurrent/distributed systems and in programming languages. The framework is applied in three steps. Firstly, individual components are wrapped in such a way that the wrappers perform activity related to local error detection and exception handling, and signal, if necessary, external exceptions outside the component. At the second step the execution of the overall system is structured as a set of dynamic actions in which components take parts. Such actions have important properties which facilitate exception handling: they are atomic, contain erroneous information and serve as recovery regions; also, actions can be nested so that systems can be developed recursively. The last step is designing exception handling at the action level: each action (i.e. all components participating in it) handles exceptions signalled by individual wrapped components.

Keywords: interface exceptions, wrappers, atomic actions, cooperative recovery

1. Exception Handling

Exception handling is a disciplined and structured way of handling abnormal system events [C95]. Exception handling features allow programmers to *declare* exceptions, to treat a program unit as the *exception context* and to associate exceptions and *exception handlers* with such context, so that when an exception is raised in this context, execution stops and a corresponding handler is searched for among the handlers (there are models in which an exception can be propagated straight outside the context).

The vital feature of any exception handling mechanism is its ability to differentiate between *internal exceptions* to be handled inside the context and the *external exceptions* which are propagated outside the context: these exceptions are not clearly separated in many languages although it is obvious that they are intended for different purposes. This separation can be done provided the following conditions are met: contexts are associated with program units which have interfaces and the concept of *context nesting* is defined. Several models clearly separate these two types of exceptions; e.g. Java, Modula-3, Coordinated Atomic (CA) actions [X98].

In the component-based system development (CBSD) interface exceptions are an immanent part of the component interfaces, which are used by component providers/developers to inform the environment that uses the component about situations in which the component cannot provide the required service. The signalled exception can be accompanied by some information about the reasons for it and about the state in which the component has been left to allow the environment to take appropriate actions.

Most existing exception handling mechanisms (e.g. in C++, Ada) use dynamic exception context nesting in which case the execution of the context can be completed either successfully or by propagating an interface exception - this exception is treated as an internal exception raised in the *containing context.* The simplest example of the dynamic nested context is nested procedure calls. Actually this is the dominating approach which suits the client/server or remote procedure call paradigms.

External exceptions allow programmers to pass (in a disciplined, unified and structured fashion) different outcomes to the containing context. This can be used to inform it of the reasons for abnormal behaviour and of the state in which the context has been left, to pass partial results, etc. Another important issue which exception handling models have to address is defining the state in which the context is left when an external exception is propagated. Some systems provide an automatic support which guarantees the "all-or-nothing" semantics: if an exception is propagated outside, all modifications made inside the context are cancelled. Another possibility (which originates in the Inscape development environment [P89]) is to allow the context to be left in several states: an initial state (an abort exception is propagated); successfully committed state (if no exception is propagated outside); and several "partial" committed states, when the requested result cannot be achieved but partial (or degraded, alternative) results are still acceptable. It is clear that developing supports to provide such functionalities is a difficult task, this is why

in many systems all responsibility of leaving the context in a known and consistent state rests entirely with application programmers.

Exception handling is a very important part of any general structuring technique used in system design as it adds new ways of concern separation which are vital for dealing with abnormal situations: it allows us to separate normal code from exception handlers during system design and structuring, introduces a dynamic separation of the execution of normal code and handlers, and provides two ways of returning the control flow after the execution of a component. Exception handling mechanisms should rely on the way the system is structured and be an integral part of system design. Many researchers regard exception handling as a means for achieving system fault tolerance [C95] and we share this view. In this context exception raising follows error detection, exception handling equals to error recovery and units of system structuring are units of exception handling and of recovery. Exception handling is used for incorporating application-specific fault tolerance. General exception handling features allow system developers to tolerate faults of various types, including environmental faults, design bugs (by employing diversity), errors reported by the underlying software or hardware levels (OS, platforms, etc.), mistakes made by operators, etc.

2. Motivations

CBSD [B00] has been a focus of research for industry and universities because it promises decreased system complexity and reduced cost of development. Components are developed to be used (and re-used) for composing new systems. Their defining characteristics are [S98]: they are units of independent deployment encapsulating their constituent features that are well separated from the environment and other components; they are units of third-party composition encapsulating their implementation and interacting with their environment through well-defined interfaces.

Developing complex systems by component integration is complicated by many factors: introducing new or extended functional and non-functional requirements (e.g., adding functionality, improving dependability), using components in a different (wider or narrower) context or environment, heterogeneity of components. There are many reasons why components may not fit well into the integration process or match each other. Component wrapping is used to overcome such problems as it addresses them without

having to modify the components themselves. Wrapping in component-based systems has many specific characteristics. First of all, general wrapping techniques can be developed for a particular implementation of components (e.g. for CORBA) because in this case all components follow the same interface agreements. Secondly, component wrappers can be easily re-used; this can improve productivity and give a possibility of separate validation of component wrappers.

CBSD is a new challenging area [B00] and many existing concepts have to be adjusted or developed further to fit its characteristics. Our focus is on developing a framework for handling exceptions in the integrated systems. Integrating components means integrating and accommodating their normal and abnormal behaviour. There is a need in disciplined exception handling for many well-known reasons (including those briefly outlined in Section 1). But, in our opinion, CSBD needs more discipline and rigor in handling exceptions, as component systems are usually very complex, they have to deal with a big variety of abnormalities without having any knowledge about the internal structure or behaviour of components.

There is clearly a need in applying *enhanced application-specific error detection* at the level of individual components. Integrators are usually reluctant to put high trust on components or their specification, this is why there is a need to develop powerful error detection features and to employ some sort of defensive programming. Moreover, the integrators usually do not have the complete specification (of both the normal and the abnormal behaviour) of the component. It is a well-known fact that the exceptional behaviour of components is always under-specified or, even, not specified [K00, S97]. In addition, we need local error detection because it is very likely that there are mistakes in components and there specifications, error detection inside components is not perfect and components are used not exactly in the contexts they are intended for.

There is a need in an *additional exception handling* that is local to each component. It allows integrators to access damage and find out the reason for the detected error, to put the component into a known consistent state, to try local error recovery and to deal with possible mismatches when components have different rules of informing the environment about exceptions raised or errors found.

The choice of the right approach to incorporating such local error detection and exception handling features is vital. Unfortunately, even though some component technologies offer structuring techniques for

developing wrappers, there is not enough attention to the problems of developing wrappers that provide error detection and exception handling.

When local handling is not possible exceptions should be propagated to the *system level* and handled there. But we believe that it would be wrong and error-prone to view the system as a flat set of all incorporated components and to leave it with the system integrator to decide which of them to involve in handling each abnormal situation. Integrated systems usually have much more complex architecture than the client/server one and integrators need general techniques applicable for structuring any complex systems during their integration to make system-level error containment and exception handling easier.

Researchers working on system dependability realise that there are many situations when it is not enough to recover only one process of complex concurrent and distributed systems [R75, C86, X95] because erroneous information can be propagated among processes, mistakes can be made in designing process joint activity; exceptions raised concurrently in several processes can be the symptoms of the same problem. This understanding is not common for CBSD (to the best of our knowledge only [D98] realises these concerns) and we intend to propose ways of applying these general techniques in the area.

We believe that problems of incorporating disciplined exception handling into CBSD has not been addressed properly, although importance of dealing with exception handling in CBSD is emphasised in [D98, S00]. Analysis of existing component technologies clearly shows that they do not provide exception handling features capable of solving problems outlined in this section.

3. Assumptions

In our model an integrated system is to be developed by deploying existing (ready-made, legacy) components treated as black boxes [B00]. The only way of using components is via these legal interfaces. System integrators do not have access to the component code nor they have any knowledge about the way the component has been implemented. Some researchers rely on different assumptions and consider that either the component developers can be asked to modify it on request [K97] or view the component as a grey box allowing an access to internal data [S97]. The assumption is that the system integrators know what the component does as there is a specification of its interface. We do not assume that it is complete and correct.

We assume that each interface uses a way of reporting abnormal events to inform the caller about the fact that the component cannot provide the required service. These can be interface exceptions, error return codes, etc. We further assume that components have deterministic behaviour and do not change their state spontaneously, so that only one exception can happen in a component at a time.

The framework should allow tolerating the following faults: design bugs in components, mistakes in the component specification and implementation, misuse of components, faults in the environment, operators' mistakes.

4. The Exception Handling Framework

Our approach relies on:

- structuring complex integrated systems out of dynamic atomic actions and associating system level exception handling with such actions
- introducing local error detection and exception handling for each component
- implementing these local functionalities using wrapping techniques.

We believe that atomic actions [C86] form the sound basis of structuring integrated systems mainly because they offer a recursive approach to building complex systems and for incorporating exception handling into them. Several participants (threads, objects, etc.) enter such an action and cooperate inside it to achieve joint goals (Figure 1). Its participants share work and explicitly exchange information in order to complete the action successfully. Actions are structuring units hiding intermediate steps of state and behaviour changing. Atomic actions structure dynamic system behaviour. To guarantee action atomicity, no information is allowed to cross the action border. Actions can be nested (a subset of the participants of the containing action can join a nested action). Participants leave the action together when all of them have completed their job. If an error is detected inside an action all participants take part in a cooperative recovery because the action is the damage area. The main reason being that processes have exchanged potentially erroneous information while performing a joint activity. Atomic actions provide a framework for developing schemes intended for tolerating faults of different types: hardware faults, software design faults, transient faults, environmental faults, etc.

Many researchers realise the importance of guaranteeing the *atomicity* of structuring unit execution. Structuring complex systems using atomic actions offers a straightforward choice of the exception contexts [R01]. Treating these units as such contexts is the most beneficial way because these atomic units have clearly defined borders, can be nested (in the same way in which exception contexts are nested) and because no information can cross the unit border. It is important that this approach is compatible with the way we structure sequential systems for exception handling, which is based on nested procedure calls. The general exception handling model can easily be applied here to allow internal exceptions and corresponding handlers to be associated with such structuring unit. Atomic actions have interfaces enriched by external exceptions which the unit can propagate into the containing exception context (i.e. into the containing structuring unit) [X98]. Atomicity of actions (i.e. of exception contexts) is vital for dealing with abnormal events (i.e. exceptions) as it guarantees the containment of all (potentially erroneous) information which should be involved in exception handling and recovery. Clearly, the atomicity of action execution has a general importance for all phases of system development [R01]: it facilitates reasoning about the system, system understanding, verification and development, tolerating faults of different types, etc.

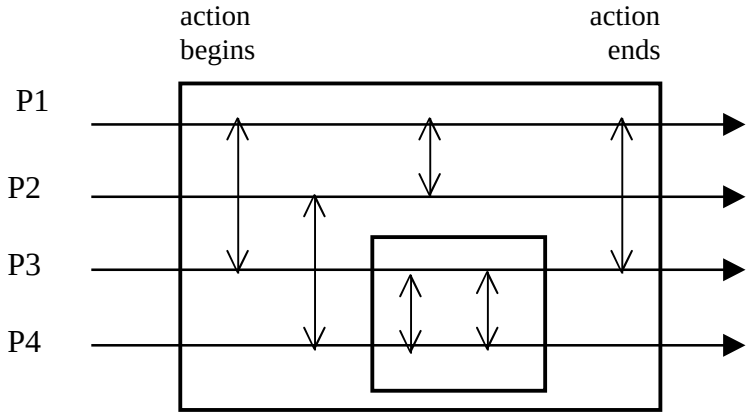

Figure 1. Atomic actions: participants P1-P4 take part in the containing action, inside which participants P3 and P4 take part in the nested action

Atomic actions is a general approach applicable for different software architectures and application areas. Our proposal is to use it in CBSD and to structure the integrated system using atomic actions in which components take part. This approach has to be adjusted for CBSD. First of all, because we employ local handling at the level of individual components. In our framework if a local handling does not succeed

an exception is propagated to the level of an action in which this component is involved to be handled cooperatively. When action level handling is not possible an exception is propagated to the containing action.

Although error detection is not usually considered to be a part of the exception handling mechanism we address it in one framework because very often the same information is used for detection and recovery, and because error detection and exception handling are essentially application-specific activities, provided by the same software associated with the same structuring units (individual components or groups of components).

In the following we show how to structurally incorporate new software providing error detection and exception handling features into a component-based system and discuss in detail CBSD-specific error detection and exception handling.

5. Local Error Detection and Exception Handling

Local error detection and exception handling, the needs for which we summarised in Section 2, are performed at the level of the standard component interface, they are "local" as they are developed for each component and do not involve other components. Although we call it "local" we consider it to be a higher level context than the internal context of the component, within which the component developers might handle or might try to handle exceptions before returning information through the component interface. Our proposal is to introduce a special "higher" (wrapper) level, which hides the component (Figure 2). This is a well know structuring mechanism and a number of implementation techniques have been developed to support it. In our context wrappers perform local error detection and exception handling.

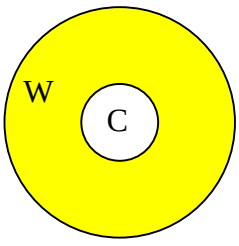

Figure 2. Component C is wrapped into newly developed layer W which intercepts all service requests directed to C and all replies

5.1. Local Error Detection

The wrappers contain errors by performing component error detection: by catching all exceptions and error return codes, and by checking predicates. These predicates are to be developed in the course of wrapper development as an important part of system integration: they describe the correct or expected behaviour of the component and are made executable. They include known restrictions on the way the component is used in the integrated system. For example, if we build an Internet travel agency system using an existing flight reservation service we might decide not to use APEX flight tickets at all as these cannot be cancelled. Another important restriction, which the system developer might decide to impose on the component execution, is that it is only allowed to use only the standard component interface: very often providers of a component offer undocumented or non-standard functionality but the integrators might decide against using it to improve compatibility.

The wrapper prevents the component from misuse by checking that calls and input parameters are correct; some parts of the interface are not used (application specific); non-standard parts of the interface are not used; and by intercepting calls which can cause known component faults (based on injection, testing, bug reports). On the output the wrapper checks correctness of outputs and, if possible, makes sure that the component is in a known correct state.

To find out what "correct" means here the system integrators develop and formalise their views on the component behaviour and specification. There are several approaches here, but unfortunately they are not general enough. Paper [S97] focuses on developing error containment wrappers suitable for a particular microkernal: this technique relies on the complete and correct specification of the component and on dealing with several known bugs detected a priori by fault injection. We agree very much with the general view on containing errors by means of wrapping expressed in [V97]: the author believes that the wrappers should limit what the components can do to the environment and what environment can do to them. But we feel that this needs further development because this paper does not offer any detailed approach to developing such wrappers. Moreover, the only approach proposed relies on using results of fault injection while implementing wrappers.

There is clearly a need in a much more general and rigorous approach. Wrapper development is a complex engineering process which has to be supported by rigorous techniques and by a clear description of the engineering steps to be undertaken. Ad hoc development is not acceptable here. Wrapper design incorporates the system integrator's view on what the component can do, should do and should not do in the integrated system. This development uses existing (but often incomplete and unreliable) knowledge of the component functionality and the application-specific knowledge about the context in which the component is to be incorporated (e.g. restrictions on the use depending on the application profiles). System integrators design contracts between the environment and the component in the form of predicates on component inputs and outputs, and, possibly, on the component state (when it is assessable through the standard interface) used for detecting latent errors. These predicates are incorporated into the component wrapper in the form of executable assertions. It is very important from our point of view to try and develop a detailed specification of the correct component behaviour but to keep it reasonably small to allow for cost-effective run-time checking. It is clear that the majority of the system integrators would be willing to live with some run-time overheads when they use a foreign piece of code in an integrated system of any level criticality.

5.2. Local Exception Handling

Local exception handling starts when either the component signals an exception (or, an error return code) or an assertion detects a violation of the specified component or environment behaviour. This handling can include another attempt to provide the required service, search for more information about the exceptions and the reasons for it, checks of the state in which the component has been left, its recovery or operations putting it into a correct known state. The wrapped component should have a set of interface exceptions which it can signal in such a way that it can give guarantees (or, attempts to give them) that the component is left in a state which corresponds to the exception being signalled. As we explained in Section 1 external (interface) exceptions allow component developers to report several outcomes to the environment. It is vital to always leave the component in a known consistent state but experience shows that this is not always the case [S97]. It is the responsibility of the wrapper to check that this has been done properly and, if it has not, to execute appropriate operations. Guaranteeing "nothing" semantics is the most useful approach, a number of the interface exceptions can have such semantics (e.g. Service_Cannot_Be_Used or

Illigal_Input_Parameters). It is important to introduce, a special *failure* interface exception to use it when the state is which the component has been left is unknown and to advice the system not to employ it without appropriate recovery.

By bracketing each operation on the component with the software performing this additional functionality the wrapper turns it into a well-defined building block which the system integrators can use. The interface exceptions are a very important part of the wrapped component: the wrapper informs the higher system level about abnormal behaviour (providing additional information about the state of the component to allow for compensation at the higher level) and passes the responsibility for recovery to the higher level if the local recovery is not possible.

Integration of complex systems requires additional activity for developing a unified exception handling policy at the system level. There are clearly *mismatches* in different exception handling models [H01]. For example, in COM all interface methods should return a status (HRESULT) indicating success or exception/failure of the method execution, which is quite different from catch and throw in Java or C++. The system integrators should define such a policy and each wrapper should follow it when signalling exceptions.

Another possible way of handling exceptions at the wrapper level is to signal an interface exception and initiate an off-line recovery (e.g. involving operators). This requires a special functionality which the wrapper can provide: delaying all requests until the component is repaired or replaced).

Local exception handling:

- incorporate damage assessment (e.g. by calling component methods and analysing information about the detected error)
- try to handle an exception locally through the standard interface of the component. Including, recovering the component, retrying the operation, moving the component into a known and consistent state (e.g. using standard abort, initialise interface operations)
- signal an exception to the environment without executing the requested component function (if the request is erroneous)
- signal all exceptions in a unified way augmenting the exceptional outcomes with additional information (e.g. component name, function name, name of the illegal parameter, etc.).

The approach proposed in this section makes exception handling cheaper because it promotes early error detection by executing assertions each time component is called, local exception handling, unification of exception propagation to the system level, leaving components in a known consistent state when an exception is propagated. To conclude the section we would like to emphasise again that we believe that errors should ideally be detected at the level of components by the wrappers and when they are detected an attempt should be made to handle them locally. If this is not possible an exception should be signalled to the environment in such a way that the component is left in a known and consistent state to facilitate the following handling at a higher level.

6. Action Oriented Structuring and Exception Handling

As we have emphasised before the integrated systems should be structured in such a way that the units of structuring are the exception contexts. Our proposal is to develop such units as atomic actions. Several wrapped components are dynamically involved in the execution of the same action (Figure 3). These actions form the dynamic structure of the integrated system as they can be recursively nested. For the system level fault tolerance it is important that such actions form the recovery areas and that all participants are involved in cooperative exception handling when any wrapped component signals an exception.

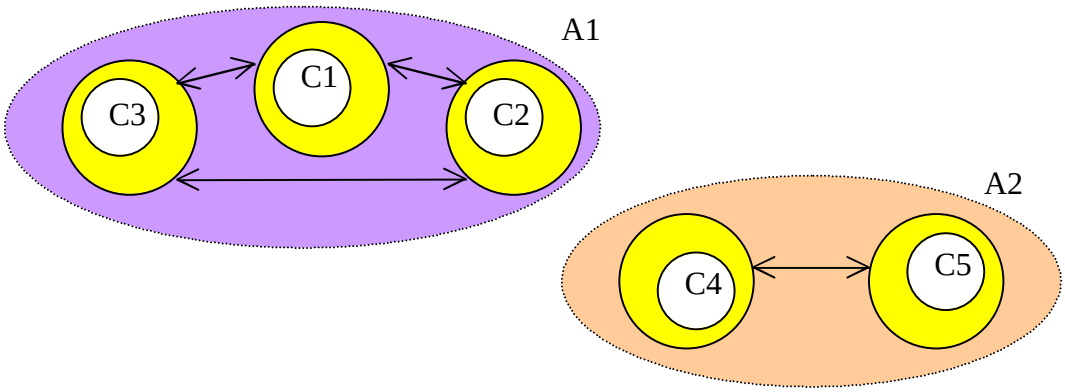

Figure 3. Dynamic actions A1 and A2. While in an action wrapped participants can cooperate with other participants (e.g. C1, C2 and C3 cooperate within A1).

During CBSD the integrators should make the decisions about the system structure including dynamic system structuring out of actions. To do the latter they have to understand the joint activities to be

performed by sets of coordinated components. The process of structuring includes, among other things, a description of the action nesting. In our opinion the importance of developing and applying special approaches to structuring system execution is not sufficiently recognised. This is a vital dimension in system development because only in this way we can introduce units of system structuring which are, in particular, exception contexts.

In real systems there are situations when complex error detection should be used to allow the integrators to check complex conditions which involve the states of a number of interrelated components. In our framework this type of error detection is applied within an atomic action. When an error is detected all participants are involved in the action level exception handling as the local exception handling cannot resolve the problem. Action level handling can include compensation for (unnecessary) updates committed by individual components or, even, for mistakenly taken actions at the component level.

Action level recovery suits well to recovering component systems because the component results are usually committed and it is not always possible to abort them, so there is a need in using system level compensation.

The approach proposed is recursive and when the action participants are not able to handle an exception they pass the responsibility for recovery to the higher system level. Handling at this level involves all components participating in the containing action. If several exceptions have been raised in an action they all have to be taken into account so that all action participants are involved in handling a resolved exception [C86].

To apply the framework proposed system integrators design cooperative component activities using nested atomic actions, describe how each individual component is involved in such activities, and develop cooperative exception handing for all participants of each actions.

7. Combining Local and Action Level Handling

In our framework the integrated system execution is structured using well-defined and wrapped individual operations on the components and (nested) actions in which components take part (Figure 4). Exceptions are handled either at the level of a component executing an individual operation or at the level of an action. If handling is not successful an interface exception is propagated to the containing action.

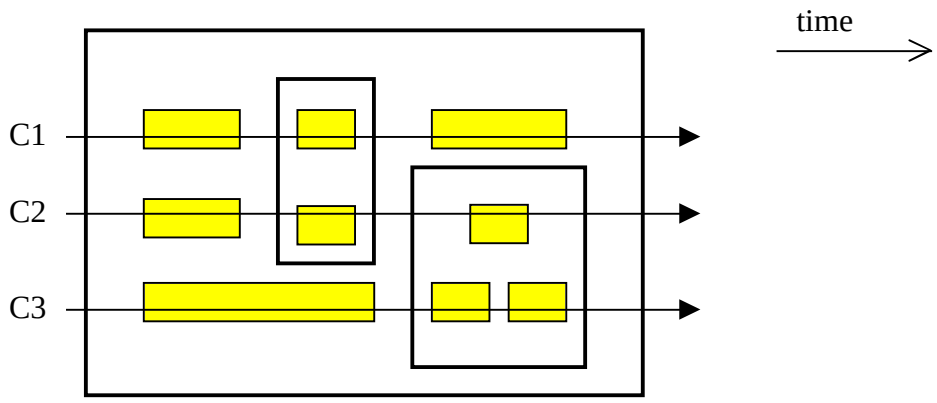

Figure 4. Execution of an integrated system with 3 components: C1, C2 and C3. Boxes with bold sizes represent atomic actions. Wrapped operations executed by individual components are shown by shadow boxes.

The order exception handling and propagation is as follows:

```
if "exception is propagated from the component" then
  if "local handler exists" then
       call a local handler;
       if "not handled" then
              go to a higher (action) level handling;
       end if;
  else
       go to a higher (action) level handling;
  end if;
else
  local error detection;
  if "exception is raised (error has been found)" then
       if local handler exists then
              call a local handler;
              if not handled then
                     go to a higher (action) level handling;
              end if;
       else
              go to a higher (action) level handling;
       end if;
  end if;
end if;
```

8. Discussion

The main purpose of the paper is to propose a general framework and to discuss engineering steps of applying it. We clearly realise the importance of implementing a support for the framework. The challenge here is to develop these functionalities for existing component technologies. Our current work focuses on developing features supporting the framework proposed in CORBA. They rely on introducing wrappers using legal ways of call interception and on adding new services which allow components to take part in atomic actions and, in particular, in a cooperative exception handling.

Local error detection and handling are implemented as component wrappers. There are many wrapping techniques. Component technologies use the ideas of wrappers to transparently insert calls to services. These are termed interceptors in CORBA and DCOM+, and proxies in CORBA3 and Enterprise JavaBeans. The CORBA2 specification allows for interceptor services that can be inserted into the normal invocation path for CORBA objects. The interceptor service is registered with the ORB that then ensures that when the client sends a request to an object the request is passed through the interceptor service and on return the result also passes through it. CORBA3 generates proxies that stand in place of the target component and allow interception of method invocations sent to the component.

A number of atomic action schemes incorporating different fault tolerance techniques have been developed for different languages: CSP, Concurrent Pascal, Ada, Java, OCCAM; for distributed, multiprocessor and single computer settings; for different application requirements. In implementing a new service for controlling actions and for cooperative exception handling within the standard component environment we rely on previous experience in designing similar services for distributed Ada and Java settings [R97, X98]. To do this we introduce an action coordinator object, one for each action, that keeps references to (or is referenced by) all action participants, as well as, to all active nested actions. New API will have to be provided to include operations of action entry and exit, as well as, exception raising to allow the coordinator to involve all action participants into handling. The activity of the individual components within the integrated system (including their participation in actions and cooperative exception handling) is described by system integrators and implemented using these services (Figure 5).

We are now working on the first case study: an Internet Travel Agency which allows users to book complete trips (including, flight, trains, renting cars, booking hotels, etc.). The system is implemented by

integration existing web services that are not aware of the fact that they participate in the execution of a bigger system. There is a number of complex issues to be addressed including providing consistency of booking in case some services cannot satisfy the requests or fails during booking, and developing wrapping techniques replacing existing ways of accessing the web services with a call interface.

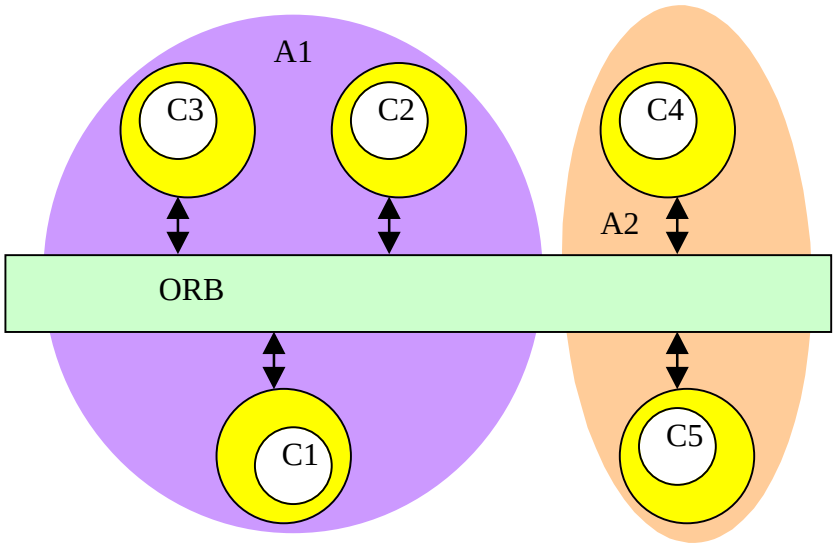

Figure 5. Component based system structuring for exception handling.

Another direction of our current and future interest lies in developing architectural styles and design patterns supporting system integration using the framework proposed [B01], that promote introducing exceptions and exception handling into earlier phases of system integration and provide new types of connectors, capturing cooperative activity of several components.

Researchers working in component-based system development have realised the needs in using a systematic approach to introducing exception handling into integrated systems [D98] but we believe that at the level of the integrated systems exceptions should be handled by applying suitable structuring techniques. Such techniques (as opposed to the techniques based on AI) allow integrators to define rigorously the exception context. Approach in [S00] is not recursive as it does not rely on any structuring and leaves all responsibility of involving several components into handling with integrators. Existing component technologies provide a unified basis and powerful features for system integration but they do not offer any systematic approaches to incorporating exception handling. Moreover the native exception

handling in these technologies is very basic (effectively the sequential one) as it relies on the client/server paradigm and is not suitable for integrating complex distributed applications.

9. Conclusions

The main ideas presented here are as follows. The paper:

- proposes a framework for introducing structured exception handling into component-based system development that relies on two level exception handling: the (local) component level and the integrated system level
- introduces local error detection as a part of the framework and discusses the CBSD-specific sources of errors to help in developing software to perform this functionality
- proposes developing local error detection and local exception handling using the existing wrapper techniques
- outlines a systematic approach to designing wrappers incorporating local error detection and exception handling
- proposes using the well-established concept of atomic actions to structure integrated systems and to incorporate exception handling at the system level in a disciplined and systematic way
- outlines possible approaches to implementing the framework support within the standard component technologies.

The introduction of disciplined and structured exception handling into CBSD is vital for building complex modern applications. Unfortunately, system integrators and researchers working in the area are often unaware of its significance. The paper attempts to put this right by making system integrators introduce exception handling starting from the earlier steps of integration and by promoting a disciplined accommodation of the existing exception handling features of individual components into integrated systems. The framework supports a systematic development of exception handling during system integration, makes this integration simpler by separating different CBSD-specific concerns and improves the overall system dependability by promoting a disciplined handling of abnormal situations.

Acknowledgements. My thanks go to Ian Welch for his help in understanding CORBA. This research is supported by European IST *Dependable Systems of Systems* and EPSRC/*UK Diversity with Off-the-Shelves Components* Projects.

References

[B00] F. Bachman, L. Bass, C. Buhman, S. Comella-Dorga, F. Long, J. Robert, R. Seacord, K. Wallnaau. Technical Concepts of Component-Based Software Engineering. TR-008. CMU. SEI (2000)

[B01] D.M. Beder, B. Randell, A. Romanovsky, C.M.F. Rubira. On Applying Coordinated Atomic Actions and Dependable Software Architectures for Developing Complex Systems. To be presented in ISORC 2001, Magdeburg (2001)

[C86] R.H. Campbell, B. Randell. Error Recovery in Asynchronous Systems. IEEE TSE-12, 8 (1986) 811-826

[C95] F. Cristian. Exception Handling and Tolerance of Software Faults. In Software Fault Tolerance, Lyu, M. (ed.). Wiley (1995) 81-107

[D98] C. Dellarocas. Toward Exception Handling Infrastructures for Component-Based Software. Position paper, Int. Workshop on Component-Based Software Engineering, Japan (1998)

[H01] B.E. Hansen, H. Fredholm. Adapting C++ Exception Handling to an Extended COM Exception Model. In Advances in Exception Handling Techniques, A. Romanovsky, C. Dony, J. Knudsen, A. Tripathi (Eds.) LNCS-2022 (2001)

[K00] P. Koopman, J. DeVale. The Exception Handling Effectiveness of POSIX Operating Systems. IEEE TSE-26, 9 (2000) 837-848

[K97] J.C. Knight, W. Lubinsky, J. McHugh, K.J. Sullivan. Architectural Approaches to Information Survivability. University of Virginia, TR CS-97-25 (1997)

[P89] D.E. Perry. The Inscape Environment, in the 11th Int. Conf. On Software Engineering. Pennsylvania (1989) 2-11

[R75] B. Randell. System Structure for Software Fault Tolerance. IEEE TSE-1 (1975) 220-232

[R01] A. Romanovsky, J. Kienzle. Action-Oriented Exception Handling in Cooperative and Competitive Concurrent Object-Oriented Systems. In Advances in Exception Handling Techniques, A. Romanovsky, C. Dony, J. Knudsen, A. Tripathi (Eds.) LNCS-2022 (2001)

[R97] A. Romanovsky, A.F. Zorzo. On Distribution of Coordinated Atomic Actions. ACM Operating Systems Review, 31, 5 (1997) 70-78

[S97] F. Salles, J. Arlat, J.-C. Fabre. Can We Rely on COTS Microkernals for Building Fault-Tolerant Systems, in the 6th Workshop on Future Trends in Distributed Computing Systems, Tunisia (1997)

[S00] M. Sparling. Lessons Learned: Through Six Years of Component-based Development. CACM. 42, 10 (2000) 47-53

[S98] C. Szyperski, Component Software - Beyond Object Oriented Programming. Addison-Wesley, 1998

[V97] J. Voas, K. Miller. Interface Robustness for COTS-based Systems, in the Colloquium COTS and Safety Critical Systems, IEE Group C1, January 1997

[X95] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery, in the 25th Int. Symp. on Fault-Tolerant Computing, California (1995) 499-509

[X98] J. Xu, A. Romanovsky, B. Randell. Coordinated Exception Handling in Distributed Object Systems: from Model to System Implementation, in the 18th. Int. Conf. on Distributed Computing Systems, Amsterdam (1998) 12-21